Homework 1 Report

Tiantu Xu

Platform:

Intel Core i7 (Quad Cores), 16GB Memory; OS: Ubuntu 16.04

APIs:

Python:

```
Conv2D(in_channel, o_channel, kernel_size, stride, mode)
[int, 3D FloatTensor] Conv2D.forward(input_image)
```

```
Image loading: PIL
Making 3D kernel: torch.stack()
Generating random kernels: torch.randn()
Tensors multiplication: torch.mul()
Save images: misc.imsave
Plot generation: matplotlib
```

C:

```
[int] c_conv(in_channel, o_channel, kernel_size, stride)
```

Images:

image0.jpg, 1280x720

image1.jpg, 1920x1080

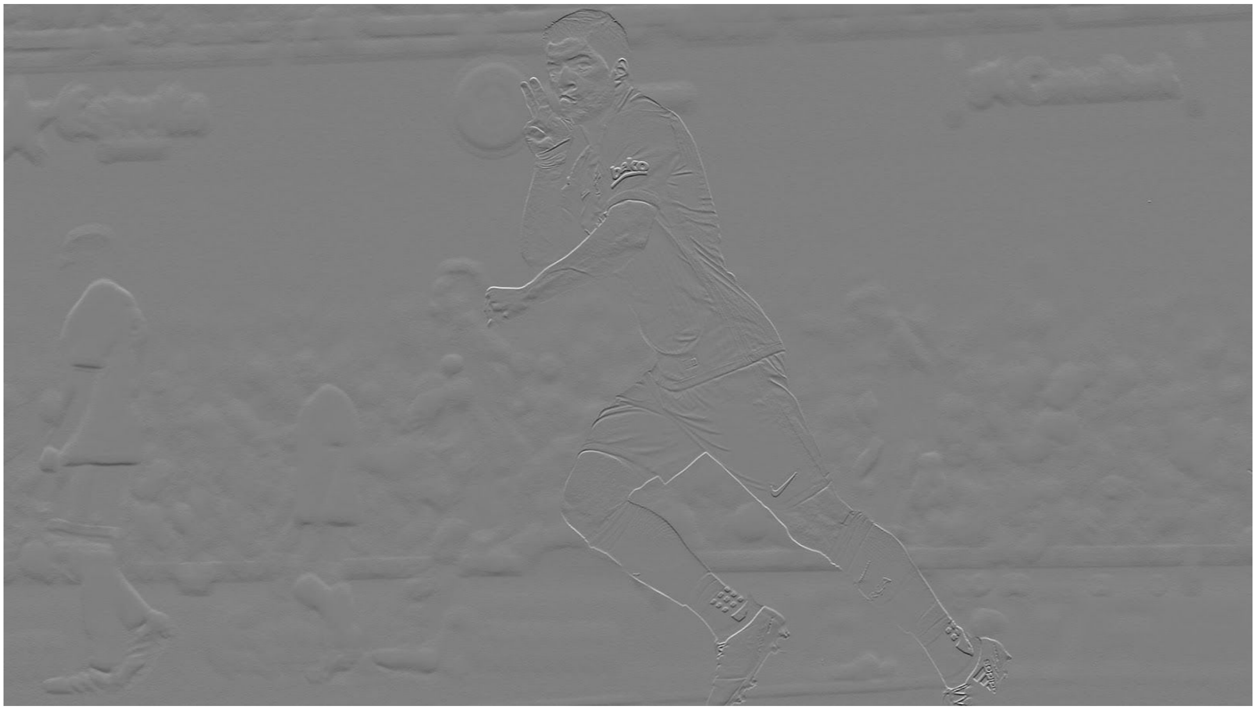
Part A:

The output grayscale images could be seen in the directory. There are 6 images for each of the input images, in total 12 images.

Task 1

1280x720

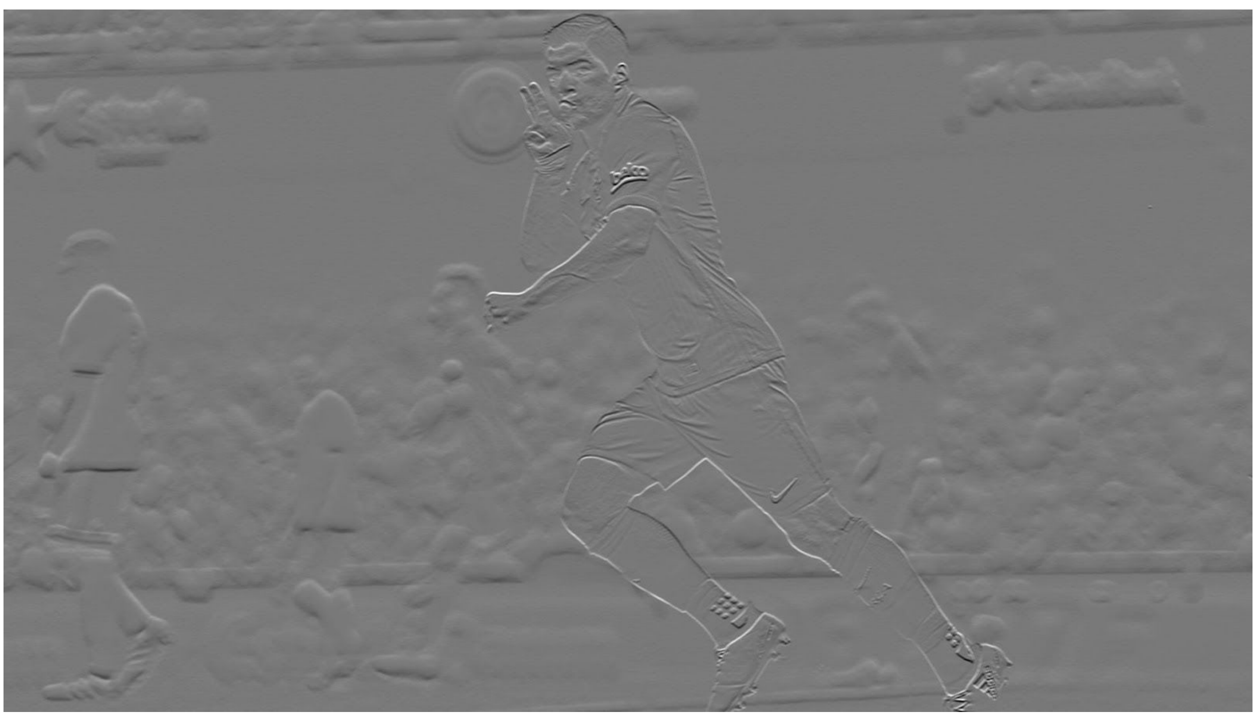
1920x1080

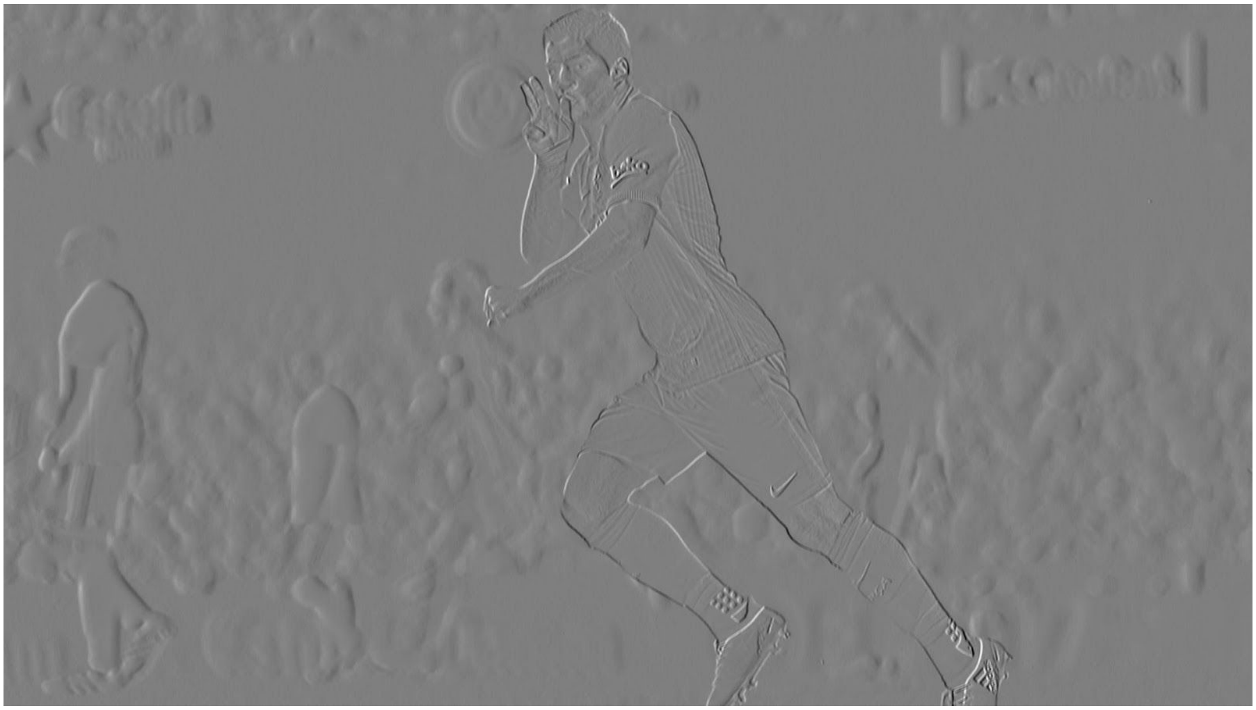


Task 2

1280x720

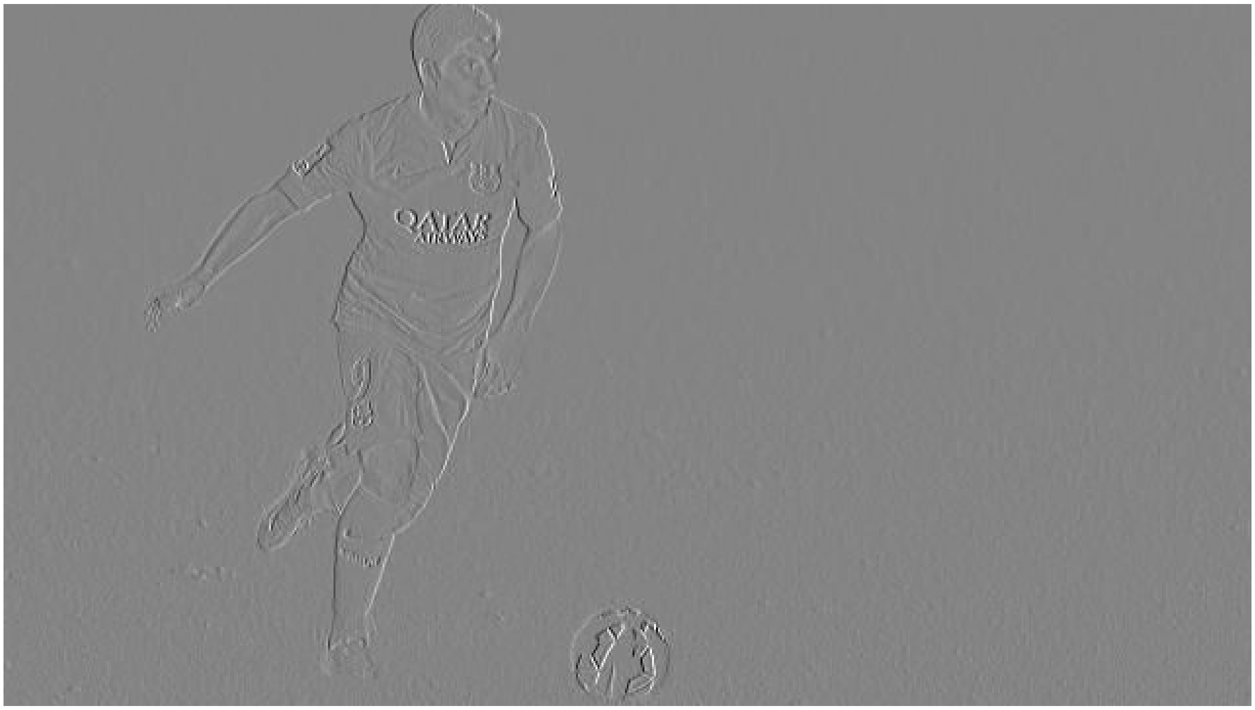
1920x1080





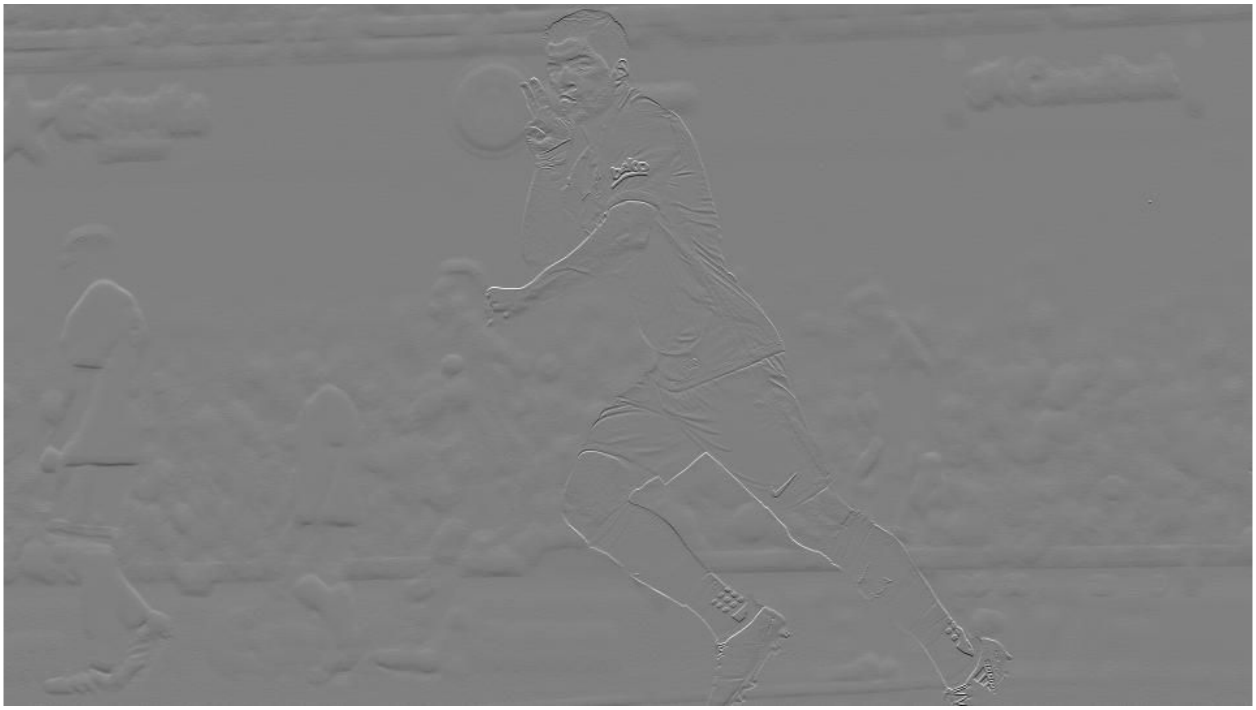
Task 3

1920x1080

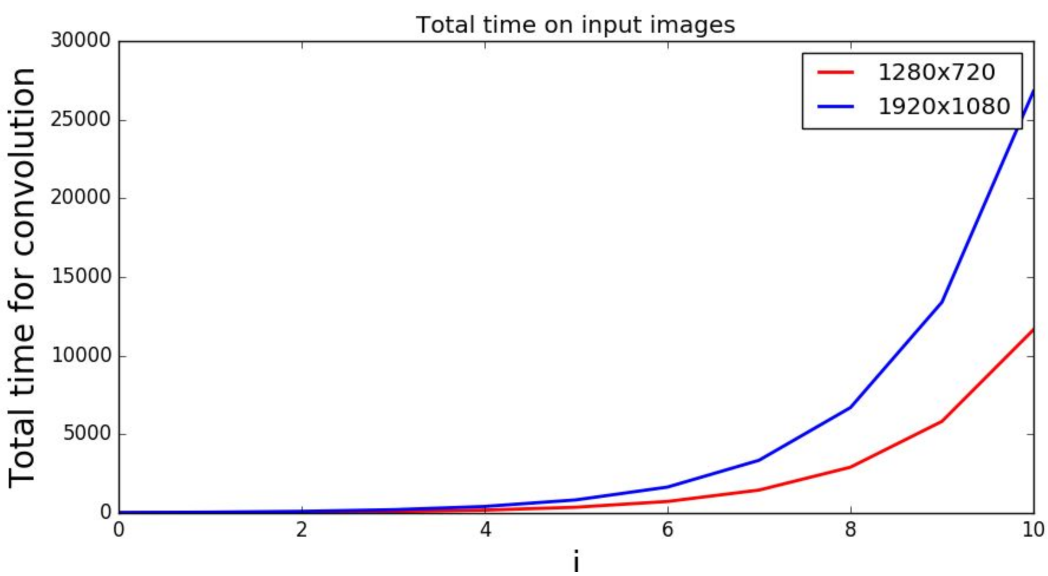

QATAR
AIRWAYS


QATAR
AIRWAYS

1920x1080

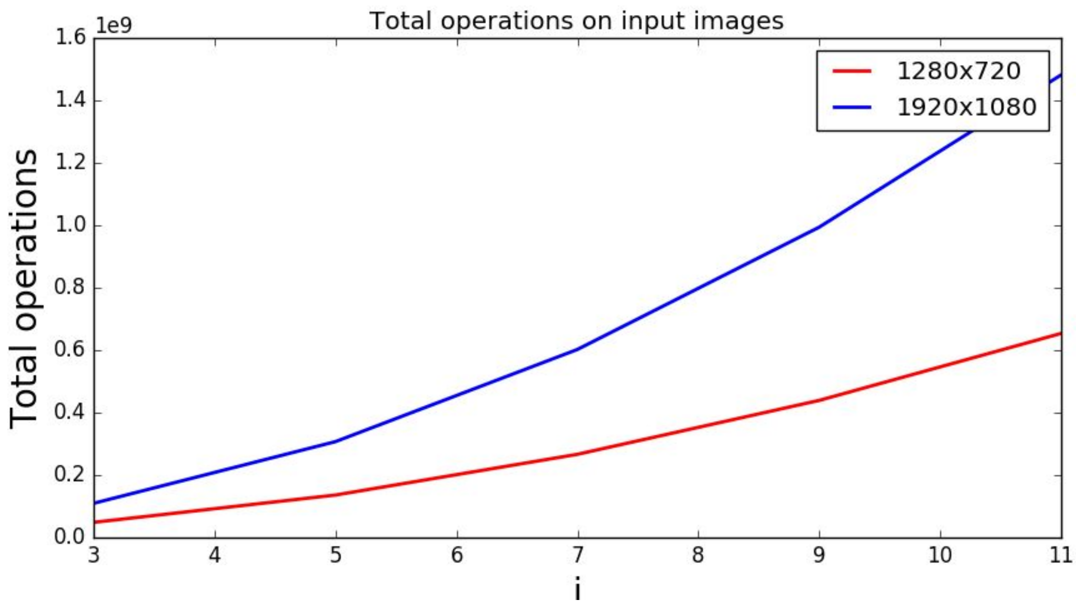


Part B:
The total time of convolutions is shown in the following graph. The red line represents 1280x720 image, while the blue line represents 1920x1080 image. 1920X1080 image apparently will take longer time to do the convolution operations. When i is larger than 7, it will take longer than 1 hour to do convolution. The time goes up with trend of 2^i.



Part C:

The The total operations of concolutions is shown in the following graph. The red line represents 1280x720 image, while the blue line represents 1920x1080 image.



To run:

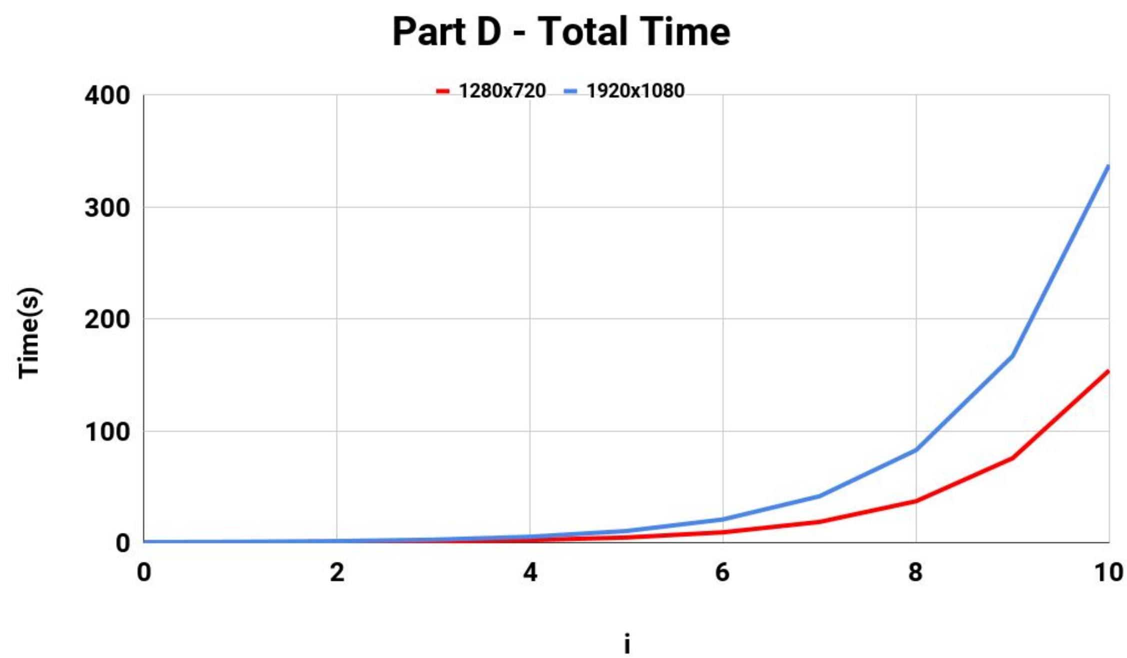
For Part A, B, C in command line

$ python main.py

Part D:

Image 0	Image 0	Image 1	Image 1
i	Time (s)	i	Time (s)
0	0.180275	0	0.390372
1	0.315048	1	0.706856
2	0.609943	2	1.338697
3	1.176244	3	2.61978
4	2.315887	4	5.233021
5	4.584614	5	10.380482
6	9.225971	6	20.748623

7	18.492344	7	41.47913
8	37.032501	8	82.688019
9	75.480141	9	166.764191
10	153.943375	10	337.67984



To Compile:

$ gcc main.c -lm -o main

To Run:

$ ./main

Source Code:

--- main.py ---

```
#!/usr/bin/env python
import numpy as np
import io
import torch
import torchvision
from conv import Conv2D
from PIL import Image
from torchvision.transforms import ToTensor
import torchvision.transforms as transforms
```

```python
from scipy import misc
import time
import matplotlib as mpl

mpl.use('Agg')
import matplotlib.pyplot as plt

toTensor = transforms.Compose([transforms.ToTensor()])
input_image = []
input_image.append(Image.open("image0.jpg"))
input_image.append(Image.open("image1.jpg"))


print("Part A")
# Part A, Task 1
print ("Task 1")
conv2d = Conv2D(in_channel=int(3), o_channel=int(1), kernel_size=int(3), stride=int(1),mode='known')
for i in range(2):
  [Number_of_ops, output_image] = conv2d.forward(toTensor(input_image[i]))
  img_name = "image" + str(i) + "_task1_k1.jpg"
  misc.imsave(img_name, output_image[:,:,0])
  print("image " + str(i) + " operations = " + str(Number_of_ops))


# Part A, Task 2
print("Task 2")
conv2d = Conv2D(in_channel=int(3), o_channel=int(2), kernel_size=int(5), stride=int(1),mode='known')
for i in range(2):
  [Number_of_ops, output_image] = conv2d.forward(toTensor(input_image[i]))
  for j in range(2):
     img_name = "image" + str(i) + "_task2_k" + str(j) + ".jpg"
     misc.imsave(img_name, output_image[:,:,j])
     print("image " + str(i) + " operations = " + str(Number_of_ops))

# Part A, Task 3
print("Task 3")
conv2d = Conv2D(in_channel=int(3), o_channel=int(3), kernel_size=int(3), stride=int(2),mode='known')
for i in range(2):
  [Number_of_ops, output_image] = conv2d.forward(toTensor(input_image[i]))
  for j in range(3):
     img_name = "image" + str(i) + "_task3_k" + str(j) + ".jpg"
     misc.imsave(img_name, output_image[:,:,j])
     print("image " + str(i) + " operations = " + str(Number_of_ops))

# Part B
print("Part B")
index = [i for i in range(11)]
fig = plt.figure(figsize=(10, 5))
```

```
for img_count in range(2):
  total_time = []
  print("image " + str(img_count))

  for i in range(11):
     conv2d = Conv2D(in_channel=3, o_channel= 2**i, kernel_size= 3, stride=1, mode='rand')
     stime = time.time()
     [Number_of_ops, output_image] = conv2d.forward(toTensor(input_image[img_count]))
     total_time.append(time.time() - stime)

     print("i = " + str(i) + ", time = " + str(total_time[i]) + ("s"))

  if(img_count == 0):
     plt.plot(index, total_time, c="red", linewidth=2.0, label='1280x720')
  else:
     plt.plot(index, total_time, c="blue", linewidth=2.0, label='1920x1080')

plt.xlabel("i", fontsize=20)
plt.ylabel("Total time for convolution", fontsize=20)
plt.title("Total time on input images")
plt.legend()
plt.savefig("Part-B")
plt.close()

# Part C
print("Part C")
index = [2*i+3 for i in range(5)]
fig = plt.figure(figsize=(10, 5))

for img_count in range(2):
  operations = []
  print("image " + str(img_count))

  for i in range(5):
     conv2d = Conv2D(in_channel=3, o_channel= 2, kernel_size= 2 * i + 3, stride=1, mode='rand')
     [Number_of_ops, output_image] = conv2d.forward(toTensor(input_image[img_count]))
     operations.append(Number_of_ops)
     print("i = " + str(i) + ", operations = " + str(Number_of_ops))

  if(img_count == 0):
     plt.plot(index, operations, c="red", linewidth=2.0, label='1280x720')
  else:
     plt.plot(index, operations, c="blue", linewidth=2.0, label='1920x1080')

plt.xlabel("i", fontsize=20)
plt.ylabel("Total operations", fontsize=20)
plt.title("Total operations on input images")
```

```
plt.legend()
plt.savefig("Part-C")
plt.close()
```

-- conv.py --

```
#!/usr/bin/env python
import numpy as np
import torch
import torchvision
from PIL import Image

class Conv2D:
  # Class
  def __init__(self, in_channel, o_channel, kernel_size, stride, mode):
      self.in_channel = in_channel
      self.o_channel = o_channel
      self.kernel_size = kernel_size
      self.stride = stride
      self.mode = mode

  def forward(self, input_image):
      self.input_image = input_image
      self.k1 = torch.tensor([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])
      self.k2 = torch.tensor([[-1,  0,  1], [-1, 0, 1], [-1, 0, 1]])
      self.k3 = torch.tensor([[ 1,  1,  1], [1, 1, 1], [1, 1, 1]])
      self.k4 = torch.tensor([[-1, -1, -1, -1, -1], [-1, -1, -1, -1, -1], [0, 0, 0, 0, 0], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1]])
      self.k5 = torch.tensor([[-1, -1, 0, 1, 1], [-1, -1, 0, 1, 1], [-1, -1, 0, 1, 1], [-1, -1, 0, 1, 1], [-1, -1, 0, 1, 1]])

      image_height = input_image.shape[1]
      image_width = input_image.shape[2]
      #print image_width, image_height

      image_row = int((image_height - self.kernel_size)/self.stride + 1)
      image_col = int((image_width - self.kernel_size)/self.stride + 1)
      output_tensor = torch.zeros((image_height - self.kernel_size)/self.stride + 1, (image_width -
self.kernel_size)/self.stride + 1, self.o_channel)


      kernel = []

      if self.mode == 'known':
         if self.o_channel == 1:
            kernel.append(torch.stack([self.k1 for i in range(self.in_channel)]))

            for k_count in range(0, self.o_channel):
               Number_of_ops = 0
```

```
            for i in range(0, image_row):
                for j in range(0, image_col):
                    out = torch.mul(kernel[k_count].float(),
                                    self.input_image[:, i * self.stride : i * self.stride + self.kernel_size,
                                    j * self.stride : j * self.stride + self.kernel_size])
                    Number_of_ops += self.kernel_size * self.kernel_size * self.in_channel
                    output_tensor[i][j] = out.sum()
                    Number_of_ops += self.kernel_size * self.kernel_size * self.in_channel - 1

            return Number_of_ops, output_tensor

    elif self.o_channel == 2:
        kernel.append(torch.stack([self.k4 for i in range(self.in_channel)]))
        kernel.append(torch.stack([self.k5 for i in range(self.in_channel)]))

        for k_count in range(0, self.o_channel):
            Number_of_ops = 0

            for i in range(0, image_row):
                for j in range(0, image_col):
                    out = torch.mul(kernel[k_count].float(),
                                    self.input_image[:, i * self.stride: i * self.stride + self.kernel_size,
                                    j * self.stride: j * self.stride + self.kernel_size])
                    Number_of_ops += self.kernel_size * self.kernel_size * self.in_channel
                    output_tensor[i][j][k_count] = out.sum()
                    Number_of_ops += self.kernel_size * self.kernel_size * self.in_channel - 1

        return Number_of_ops, output_tensor
    else:
        kernel.append(torch.stack([self.k1 for i in range(self.in_channel)]))
        kernel.append(torch.stack([self.k2 for i in range(self.in_channel)]))
        kernel.append(torch.stack([self.k3 for i in range(self.in_channel)]))
        for k_count in range(0, self.o_channel):
            Number_of_ops = 0

            for i in range(0, image_row):
                for j in range(0, image_col):
                    out = torch.mul(kernel[k_count].float(),
                                    self.input_image[:, i * self.stride: i * self.stride + self.kernel_size,
                                    j * self.stride: j * self.stride + self.kernel_size])
                    Number_of_ops += self.kernel_size * self.kernel_size * self.in_channel
                    output_tensor[i][j][k_count] = out.sum()
                    Number_of_ops += self.kernel_size * self.kernel_size * self.in_channel - 1

        return Number_of_ops, output_tensor
else:
    for out_count in range(0, self.o_channel):
        rand_kernel = torch.randn(self.in_channel, self.kernel_size, self.kernel_size)
```

```python
        Number_of_ops = 0

        for i in range(0, image_row):
            for j in range(0, image_col):
                out = torch.mul(rand_kernel.float(),
                                self.input_image[:, i * self.stride: i * self.stride + self.kernel_size,
                                j * self.stride: j * self.stride + self.kernel_size])
                Number_of_ops += self.kernel_size * self.kernel_size * self.in_channel
                output_tensor[i][j][out_count] = out.sum()
                Number_of_ops += self.kernel_size * self.kernel_size * self.in_channel - 1

    return Number_of_ops, output_tensor
```

-- main.c --

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

long double c_conv(int in_channel, int o_channel, int kernel_size, int stride) {
  float ***kernel = (float***)malloc(in_channel*sizeof(float**));
  float ***out_array = (float***)malloc(o_channel*sizeof(float***));

  int Num_of_ops = 0;

  int i, j, k;
  int c1, c2, c3;

  static int rows = 720;
  static int columns = 1280;
  //static int rows = 1080;
  //static int columns = 1920;

  float ***input_image;

  input_image = (float***)malloc(in_channel*sizeof(float**));
  for (i = 0; i < in_channel; i++)
    input_image[i] = (float**)malloc(rows*sizeof(float*));

  for (i = 0; i < in_channel; i++)
    for (j = 0; j < rows; j++)
      input_image[i][j] = (float*)malloc(columns*sizeof(float));

  // Create input test image
  for(i = 0; i < in_channel; i++){
    for(j = 0; j < rows; j++){
      for(k = 0; k < columns; k++){
```

```
            input_image[i][j][k] = rand()%255;
        }
    }
}

// Initialize a 3D kernel
for (i = 0; i < in_channel; i++){
    kernel[i] = (float**)malloc(kernel_size*sizeof(float*));
}


for (i = 0; i < in_channel; i++){
    for (j = 0; j < kernel_size; j++){
        kernel[i][j] = (float*)malloc(kernel_size*sizeof(float));
    }
}

for(i = 0; i < in_channel; i++){
    for(j = 0; j < kernel_size; j++){
        for(k = 0; k < kernel_size; k++){
            kernel[i][j][k] = (float)(rand()%100-50.0)/50.0;
        }
    }
}

int out_rows = (int)((rows - kernel_size)/stride + 1);
int out_columns = (int)((columns - kernel_size)/stride + 1);

// Initialize a output tenor
for (i = 0; i < o_channel; i++){
    out_array[i] = (float**)malloc(out_rows*sizeof(float*));
}


for (i = 0; i < o_channel; i++){
    for (j = 0; j < out_rows; j++){
        out_array[i][j] = (float*)malloc(out_columns*sizeof(float));
    }
}

for(i = 0; i < o_channel; i++)
    for(j = 0; j < out_rows; j++)
        for(k = 0; k < out_columns; k++)
            out_array[i][j][k] = 0;

// Convolutions
for(i = 0; i < o_channel; i++){
    for(j = 0; j < out_rows; j++){
```

```c
        for(k = 0; k < out_columns; k++){
            int sum = 0;
            for(c1 = 0; c1 < in_channel; c1++){
                for(c2 = 0; c2 < kernel_size; c2++){
                    for(c3 = 0; c3 < kernel_size; c3++){
                        sum += kernel[c1][c2][c3] * input_image[c1][j+c2][k+c3];
                        Num_of_ops += 2;
                    }
                }
            }
            out_array[i][j][k] = sum;
        }
    }
  }

  return Num_of_ops;
}

int main(){
  int in_channel = 3;
  float total_time[11];
  int stride = 1;
  int kernel_size = 3;
  int i, j, k;
  long double Num_of_ops;

  clock_t start, end;

  for(int i = 0; i < 11; i++) {
      start = clock();
      Num_of_ops = c_conv(in_channel, pow(2,i), kernel_size, stride);
      end = clock();
      total_time[i] = (double)(end - start) / CLOCKS_PER_SEC;
      printf("For i = %d, computation_time = %lf \n", i, total_time[i]);
  }

  return 0;
}
```